flowTorch - a Python library for analysis and reduced-order modeling of fluid flows

Andre Weiner[1] **and Richard Semaan**[1]

1 Technische Universität Braunschweig, Institute of Fluid Mechanics, Flow Modeling and Control Group

DOI: 10.21105/joss.03860

Software

- Review
- Repository
- Archive

Editor: Kevin M. Moerman

Reviewers:

- @akaptano
- @JaroslavHron
- @salrm8
- @hkjeldsberg

Submitted: 18 October 2021
Published: 13 December 2021

Summary

The `flowTorch` library enables researchers to access, analyze, and model fluid flow data from experiments or numerical simulations. Instead of a black-box end-to-end solution, `flowTorch` provides modular components allowing to assemble transparent and reproducible workflows with ease. Popular data formats for fluid flows like OpenFOAM, VTK, or DaVis may be accessed via a common interface in a few lines of Python code. Internally, the data are organized as PyTorch tensors. Relying on PyTorch tensors as primary data structure enables fast array operations, parallel processing on CPU and GPU, and exploration of novel deep learning-based analysis and modeling approaches. The `flowTorch` packages also includes a collection of Jupyter notebooks demonstrating how to apply the library components in a variety of different use cases, e.g., finding coherent flow structures with modal analysis or creating reduced-order models.

Statement of need

Thanks to the increased processing power of modern hardware, fluid flow experiments as well as numerical simulations are producing vast amounts of highly resolved, complex data. Those data offer great opportunities to optimize industrial processes or to understand natural phenomena. As modern datasets continue to grow, post-processing pipelines will be increasingly important for synthesizing different data formats and facilitating complex data analysis. While most researchers prefer simple text-encoded comma-separated value (CSV) files, big datasets require special binary formats, such as HDF5 or NetCDF. If the data are associated with a structured or an unstructured mesh, VTK files are a popular choice. Other simulation libraries for fluid flows, like OpenFOAM, organize mesh and field data in custom folder and file structures. On the experimental side, software packages like DaVis allow exporting particle image velocimetry (PIV) snapshots as CSV files. Reading CSV files can be a daunting task, too. A sequence of snapshots might be organized in one or multiple files. If the data are stored in a single file, the file must be read first and then the individual snapshots must be extracted following some initially unknown pattern. If the data are spread out over multiple files, the time might be encoded in the file name, but it could be also the case that the files are located in individual folders whose names encode the time. The latter structure is typical for OpenFOAM run time post-processing data. Moreover, different software packages will create different file headers, which may have to be parsed or sometimes ignored. CSV, VTK, or OpenFOAM data may come as binary or text-encoded files. This list is by no means comprehensive in terms of available formats and presents only the tip of the iceberg.

A common research task may be to compare and combine different data sources of the same fluid flow problem for cross-validation or to leverage each source's strengths in different kinds of analysis. A typical example would be to compare or combine PIV data with sampled planes extracted from a numerical simulation. The simulation offers greater details and additional field information, while the PIV experiment is more trustworthy since it is closer to the real application. The PIV data may have to be processed and cleaned before using it in consecutive analysis steps. Often, significant research time is spent on accessing, converting, and processing the data with different tools and different formats to finally analyze the data in yet another tool. Text-encoded file format might be convenient at first when exchanging data between tools, but for large datasets the additional conversion is unsuitable.

`flowTorch` aims to simplify access to data by providing a unified interface to various data formats via the subpackage `flowtorch.data`. Accessing data from a distributed OpenFOAM simulation is as easy as loading VTK or PIV data and requires only a few lines of Python code. All field data are converted internally to PyTorch tensors (Paszke et al., 2019). Once the data are available as PyTorch tensors, further processing steps like scaling, clipping, masking, splitting, or merging are readily available as single function calls. The same is true for computing the mean, the standard deviation, histograms, or quantiles. Modal analysis techniques, like dynamic mode decomposition (DMD)(Kutz et al., 2016; Schmid, 2010) and proper orthogonal decomposition (POD)(Brunton & Kutz, 2019; Semaan et al., 2020), are available via the subpackage `flowtorch.analysis`. The third subpackage, `flowtorch.rom`, enables adding reduced-order models (ROMs), like cluster-based network modeling (CNM)(Fernex et al., 2021), to the post-processing pipeline. Computationally intensive tasks may be offloaded to the GPU if needed, which greatly accelerates parameter studies. The entire analysis workflow described in the previous section can be performed in a single ecosystem sketched in Figure 1. Moreover, re-using an analysis pipeline in a different problem setting is straightforward.

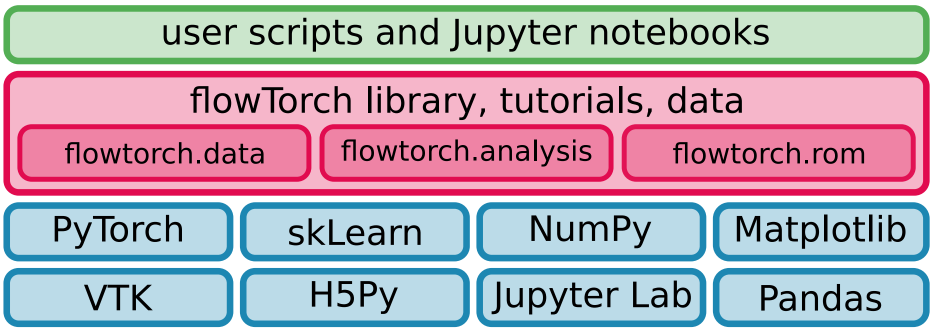

Figure 1: Components of flowTorch and library dependencies.

Besides the subpackages already available in `flowTorch`, the library also integrates nicely with related software packages like ParaView or VisIt for mesh-based post-processing as well as specialized analysis and modeling packages like PyDMD (Demo et al., 2018), PySINDy (Silva et al., 2020), or modred. Rather than re-implementing functionality already existing in other established libraries, `flowTorch` wraps around them to simplify their usage and streamline the overall post-processing pipeline. For example, we use ParaView's vtk package to access various types of VTK files in Python. Gathering point coordinates, write times, or snapshots from several VTK files requires very different steps than when dealing with OpenFOAM or DaVis data. However, due to the common interface to data sources in `flowTorch`, these tasks appear to be exactly the same for the user. In contrast to `flowTorch`, PyDMD offers a wide range of DMD variants but does not provide access to data. If an advanced DMD algorithm is required, our library can be used to access and pre-process a dataset, before PyDMD is used to perform the modal decomposition.

Another more general issue we want to address is the reproducibility of research outcomes. Popular algorithms, like POD or DMD, may be relatively easy to implement with libraries like NumPy, SciPy, or PyTorch. However, applying these algorithms to real datasets typically requires several pre-processing steps, like cropping, clipping, or normalizing the data, and careful tuning of the algorithms' free parameters (hyperparameters). Therefore, it is often unclear which exact steps were taken to produce the reported results and how robust the results are to changes in the free parameters or the data. Even if the authors are willing to provide more details, essential information may not be accessible due to black-box (closed-source) analysis tools used somewhere in the process.

With `flowTorch`, we attempt to make analysis and modeling workflows accessible, streamlined, and transparent in several ways:

- we provide Jupyter notebooks with start-to-end workflows, including short explanations for each step taken in the process; the notebooks' content varies from toy examples through common benchmark problems to the analysis of real turbulent flow data; the datasets used in the notebooks are also part of the library
- the library is modular and often wraps around other libraries to make them easier to use; a few lines of Python code are sufficient to implement a basic workflow; the modular structure and the rich documentation of the source code simplify writing extensions and enable quick automated experimentation

Ultimately, our goal is to reduce redundant work as much as possible and enable users to focus on what matters - understanding and modeling flow dynamics.

Examples

In this section, we demonstrate two applications of `flowTorch`. In the first example, DMD is employed to identify relevant modes in a transonic flow displaying shock-boundary-layer interactions. In the second example, a ROM of the flow past a circular cylinder (Noack et al., 2003) is constructed employing CNM (Fernex et al., 2021). Both examples are also available as Jupyter notebooks and in the `flowTorch` documentation.

DMD analysis of airfoil surface data

For this example, we need only a handful of `flowTorch` components.

```
import torch as pt
from flowtorch import DATASETS
from flowtorch.data import CSVDataloader, mask_box
from flowtorch.analysis import DMD
```

`DATASETS` is a dictionary holding names and paths of all available datasets. The `CSVDataloader` provides easy access to the data, and the `mask_box` function allows selecting only a spatial subset of the raw data. As the name suggests, the `DMD` class enables us to perform a DMD analysis.

The dataset we use here consists of surface pressure coefficient distributions sampled over a NACA-0012 airfoil in transonic flow conditions. The OpenFOAM configuration files to produce the dataset are available in a separate GitHub repository. At a Reynolds number of $Re = 10^6$, a Mach number of $Ma = 0.75$ and $\alpha = 4°$ angle of attack, the flow displays a so-called shock buffet on the upper side of the airfoil. The shock buffet is a self-sustained unsteady interaction between the shock and the boundary layer separation. Our aim is to extract flow structures (modes) associated with the buffet phenomenon.

A code snippet to read the data, mask part of it, and build the data matrix reads:

```
...
path = DATASETS["csv_naca0012_alpha4_surface"]
loader = CSVDataloader.from_foam_surface(
    path, "total(p)_coeff_airfoil.raw", "cp")
vertices = loader.vertices
vertices /= (vertices[:, 0].max() - vertices[:, 0].min())
mask = mask_box(vertices,
    lower=[-1.0, 0.0, -1.0], upper=[0.9999, 1.0, 1.0])
points_upper = mask.sum().item()
data_matrix = pt.zeros((points_upper, len(times)), dtype=pt.float32)
for i, time in enumerate(times):
    snapshot = loader.load_snapshot("cp", time)
    data_matrix[:, i] = pt.masked_select(snapshot, mask)
```

The `CSVDataloader` has a class method designed to read raw sample data created by OpenFOAM simulations. Every `Dataloader` implementation provides access to one or multiple snapshots of one or multiple fields and the associated vertices. The airfoil coordinates are typically normalized with the chord length, which is the difference between largest and smallest value of the x-coordinate in the present example. The data contain pressure coefficients from both upper and lower surfaces, so we create a spatial mask to extract values from the upper surface. The DMD expects a data matrix as input whose columns are individual snapshots. Therefore, we allocate a new 2D tensor with as many rows as selected points and as many columns as selected snapshots (the data loader also provides access to the available write time - not shown here). Finally, we loop over the snapshot times and fill the data matrix.

Creating a new `DMD` instance automatically performs the mode decomposition based on the provided input. We can analyze the obtained spectrum and the associated modes. The modes have real and imaginary parts, which are equally important for the reconstruction of the flow field. It is usually enough to visualize either real or imaginary part for the physical interpretation of modes.

```
dmd = DMD(data_matrix, dt, rank=200)
amplitudes = dmd.amplitudes
frequencies = dmd.frequency
modes_real = dmd.modes.real
```

In contrast to POD, the DMD modes are not sorted by their variance, but rather form a spectrum. Figure 2 presents the real part of three spatial modes with the largest amplitudes. Also shown is their corresponding frequency.

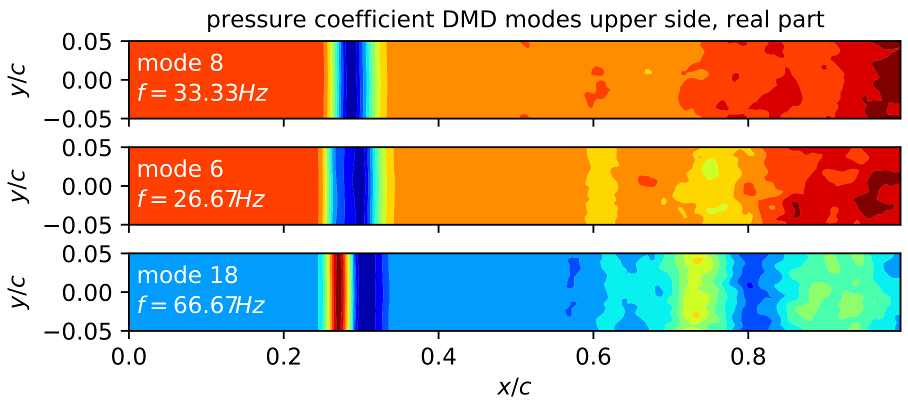

Figure 2: Real part of three dominant DMD modes over the upper surface of a NACA-0012 airfoil. The modes are normalized to the range $[0, 1]$. The coordinates are normalized with the chord c. The shock is located at $x/c \approx 0.25$. Modes 8 and 18 are harmonics. The motion of the shock front is correlated with changes in the pressure values close to the trailing edge. This effect can be nicely observed via the mode animations in the documentation and indicates the existence of a physical link between both effects.

CNM of the flow past a circular cylinder

This example demonstrates how to model a flow using the CNM algorithm (Fernex et al., 2021). Compared to the original CNM implementation available on GitHub, the version in `flowTorch` is refactored, more user-friendly, and extendible. In `flowTorch`, creating a ROM always consists of three steps: i) encoding/reduction, ii) time evolution, and iii) decoding/reconstruction. In the code snippet below, we use an encoder based on the singular value decomposition (SVD) to reduce the dimensionality of the original snapshot sequence, and then predict the temporal evolution and reconstruct the flow over the period of $1s$.

```
...
from flowtorch.rom import CNM, SVDEncoder
# load data
...
encoder = SVDEncoder(rank=20)
info = encoder.train(data_matrix)
reduced_state = encoder.encode(data_matrix)
cnm = CNM(reduced_state, encoder, dt, n_clusters=20, model_order=4)
prediction = cnm.predict(data_matrix[:, :5], end_time=1.0, step_size=dt)
```

The `predict` function computes the temporal evolution in the reduced state space and automatically performs the reconstruction. If we are only interested in the phase space, we can use `predict_reduced` instead, and reconstruct selected states using the encoder's `decode` method. The temporal evolution in the phase-space is displayed in Figure 3.

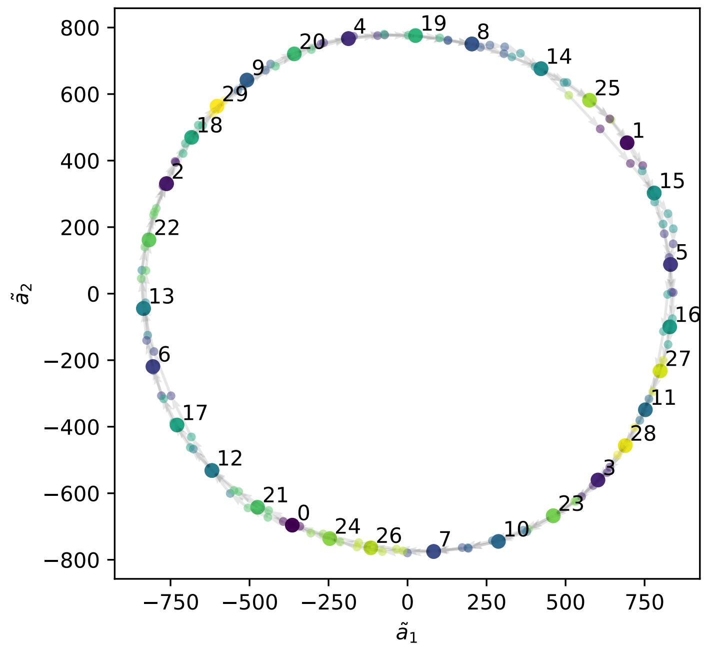

Figure 3: Phase-space representation of data clustering (large dots) and trajectory; the numbering reflects the sequence in which the centroids are visited; the smaller dots mark interpolated time steps between the centroids and are colored by their cluster affiliation (only for visualization).

Acknowledgements

The authors gratefully acknowledge financial support by the German Research Foundation (DFG) received within the research unit FOR 2895 *Unsteady flow and interaction phenomena at high speed stall conditions*.

References

Brunton, S. L., & Kutz, J. N. (2019). Singular value decomposition (SVD). In *Data-driven science and engineering: Machine learning, dynamical systems, and control* (pp. 3–46). Cambridge University Press. https://doi.org/10.1017/9781108380690.002

Demo, N., Tezzele, M., & Rozza, G. (2018). PyDMD: Python Dynamic Mode Decomposition. *The Journal of Open Source Software*, *3*(22), 530. https://doi.org/https://doi.org/10.21105/joss.00530

Fernex, D., Noack, B. R., & Semaan, R. (2021). Cluster-based network modelingfrom snapshots to complex dynamical systems. *Science Advances*, *7*(25). https://doi.org/10.1126/sciadv.abf5006

Kutz, J. N., Brunton, S. L., Brunton, B. W., & Proctor, J. L. (2016). *Dynamic mode decomposition*. Society for Industrial; Applied Mathematics. https://doi.org/10.1137/1.9781611974508

Noack, B. R., Afanasiev, K., Morzyński, M., Tadmor, G., & Thiele, F. (2003). A hierarchy of low-dimensional models for the transient and post-transient cylinder wake. *Journal of Fluid Mechanics*, *497*, 335–363. https://doi.org/10.1017/S0022112003006694

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Schmid, P. J. (2010). Dynamic mode decomposition of numerical and experimental data. *Journal of Fluid Mechanics*, *656*, 5–28. https://doi.org/10.1017/S0022112010001217

Semaan, R., Fernex, D., Weiner, A., & Noack, B. R. (2020). *xROM: A toolkit for reduced-order modeling of fluid flows* (R. Semaan & B. R. Noack, Eds.; First Edition, Vol. 1). Technische Universität Braunschweig. https://doi.org/10.24355/dbbs.084-202007011404-0

Silva, B. de, Champion, K., Quade, M., Loiseau, J.-C., Kutz, J., & Brunton, S. (2020). PySINDy: A python package for the sparse identification of nonlinear dynamical systems from data. *Journal of Open Source Software*, *5*(49), 2104. https://doi.org/10.21105/joss.02104